Kalle Laukkanen

Analysis and Example Implementation of Reusable Software Components on Small Embedded Devices

Metropolia University of Applied Sciences
Bachelor of Engineering
Information Technology
Thesis
4th December 2012

Metropolia

Tiivistelmä

Tekijä Otsikko Sivumäärä Aika	Kalle Laukkanen Pienille sulautetuille laitteille tarkoitettujen uudelleenkäytettävien ohjelmistokomponenttien analyysi ja esimerkkitoteutus 29 sivua 4.12.2012
Tutkinto	insinööri (AMK)
Koulutusohjelma	tietotekniikka
Suuntautumisvaihtoehto	sulautetut järjestelmät
Ohjaajat	lehtori Kimmo Sauren Development Manager Petri Sairila

Vaisala Oyj pyrkii parantamaan sen pieniin sulautettuihin laitteisiin tarkoitetun lähdekoodin uudelleenkäyttöä. Tällä hetkellä yrityksellä ei ole olemassa jaettua lähdekoodikirjastoa, jota useat eri kehittäjät voisivat järjestelmällisesti käyttää, vaikka useat laitteet tarvitsevatkin samanlaisia toimintoja ohjelmistoiltaan. Tämän opinnäytetyön tarkoituksena on toimia esimerkkinä kirjastonluomisprosessista toteuttamalla tarkoitukseen soveltuva ohjelmistokirjasto mahdollisimman pienessä mittakaavassa. Työtä varten valittiin pieni määrä Vaisala Oyj:n tuotteita ja kaksi näille yhteistä ohjelmistokomponenttia. Nämä komponentit analysoitiin ja näistä muodostettiin kaksi kirjastokomponenttia, joiden on tarkoitus toimia kaikissa laitteissa.

Työ koostui Vaisala Oyj:n ohjelmistosuunnittelijoiden haastattelemisesta, ohjelmistokomponenttien valinnasta ja analysoinnista, universaalien komponenttien määrittelystä ja luonnista ja ylläpitosääntöjen luonnista.

Työn lopputuloksena on kirjastoa varten kaksi ohjelmistokomponenttia, joita voidaan käyttää joko suoraan tai pienin muutoksin Vaisalan nykyisissä ja tulevissa tuotteissa. Tämän lisäksi työ tarjoaa tietoa kirjaston toteutusprosessista, sen haastavuudesta ja universaalien ohjelmistokomponenttien erilaisista vaatimuksista.

Avainsanat	Vaisala, sulautetut järjestelmät, ohjelmistokirjasto, parametrien käsittely

Metropolia

Abstract

Author(s) Title Number of Pages Date	Kalle Laukkanen Analysis and example implementation of reusable software components on small embedded devices 29 pages 4th December 2012
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Embedded Systems
Instructor(s)	Petri Sairila, Development Manager Kimmo Sauren, Senior Lecturer

Vaisala Oyj is planning on improving on their software reuse policies for small embedded devices. Currently, there exists no common library for a number of software components, despite the fact that many different devices use these same functionalities. The purpose of the project is to demonstrate the library-making process by selecting a small number of devices from Vaisala products, and two common functionalities, to get a view of the difficulty of the process. The implementation of the selected functionalities in the devices was systematically analyzed, and based on this, two universal components were created.

The project consisted of collecting information about the software components in different devices by interviewing the software designers, selecting suitable software components for the project, analyzing the selected components, assembling the universal components and establishing rules to maintain the components and establishing rules to update the components. The results of the project give information about the difficulty of the library implementation process, information about the universal component criteria and a basic guideline about the process.

Keywords	Vaisala, embedded systems, software library, parameter handling

Table of contents

1 Introduction

When writing software, there are many ways to avoid doing the same work multiple times by reusing the produced code in different ways. Sometimes, complete functionalities can be used in many different programs. Because of this, the best way for an organization to develop software is to share all the software in a depository that all the programmers have access to. However, to be useful, this would require a lot of organising and resources, and co-operation between the programmers. Currently, many programmers work independently, with no common libraries for potentially reusable software components nor rules to maintain such libraries.

Vaisala Oyj is planning on improving on the reusability of their small embedded systems software. Currently each developer has created their software on their own, causing that many functionalities are created multiple times, and sometimes behaving differently in some situations. This could be solved with a common library including the shared functionalities, and a set of rules to maintain the library.

The purpose of this project was to test the implementation process of the library in a small scale. This was done by analyzing the source code of a small number of devices selected by Vaisala software designers, and then selecting two suitable functionalities for further analysis. With the gained information, two universal components were created to suit the needs of all the analyzed devices. Also, rules for maintaining the created library were established.

When finished, the project will optimally provide Vaisala Oyj with two useful software components which can be used without modifications in a majority of devices needing the components in question. Additionally, the thesis will provide information about the difficulty and length of the library implementation process, ultimately determining if establishing of a common library is possible in a practical way.

2 Scope of the Project

The project consisted of four phases: information gathering, source code analysis, library implementation and testing, and maintenance rule establishment. They can be seen in

Figure 1 - Work phases and results

which roughly describes the work and results associated with each phase: The results after each phase are on green and the work associated with each phase is on yellow background.

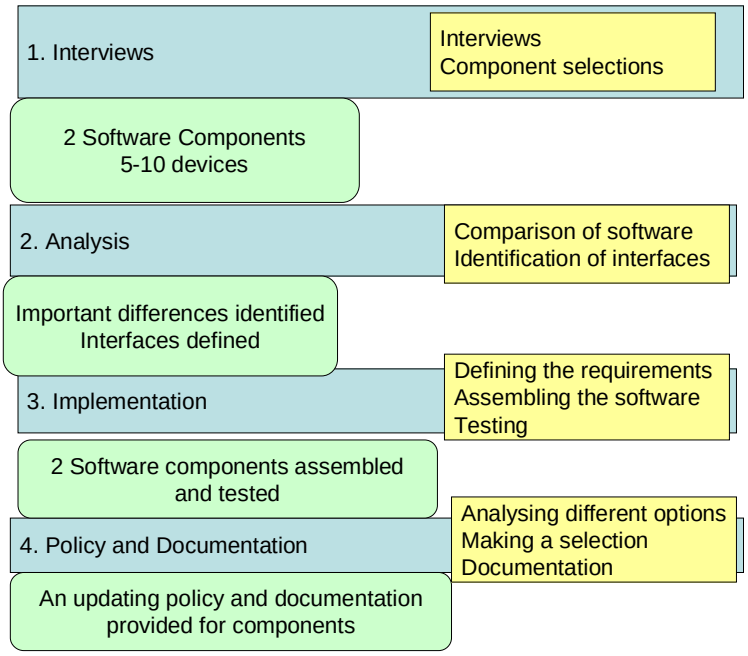

Figure 1 - Work phases and results

In the first phase, six Vaisala developers were interviewed to gain information about the devices and software components needed in the project. The developers selected five products which were used in this project, and they provided information about different software components. This information was then compared and analyzed, and based on this, two software components were selected for the library. The qualities estimated for this selection included difficulty of implementation, size of the component and frequency of appearance in devices.

At the analysis phase, the implementation of the software components on different devices was compared and analyzed. The purpose of this was to determine the requirements and limitations for the common component, and to compare different implementation methods. In practice, the analysis was done by selecting a well written source code as the base, analyzing the general structure, and then comparing the other devices to this.

Based on the analysis, the common library component was outlined. This included pointing out the structure and based on this, the rules for updating the component, establishing the interfaces and correct use and explaining all the functionality provided by the library component. Based on this, the components were assembled from the existing source code, modified as needed, and tested. Then the source code was reviewed by Vaisala software developers and updated when needed.

The final phase was to establish the policy for library maintenance. When the project was completed it provided Vaisala with strict rules on who can update the library, and how they must document the update. The final phase also included the completion of the documentation process, which consists of this document, and a how-to-use document for each software component (not provided here).

3 Gathering Information

The project started with a series of interviews with Vaisala software designers. These interviews were used to gain information about different devices in the scope of this project and about different software components suitable for the project. Five devices were selected for the project and five software components were selected for a preliminary analysis. This chapter describes the devices and software components, and givesinsight about the component selection.

3.1 Devices

3.1.1 XMW90

XMW90 is a generic piece of software for different devices, for example the wall-mounted humitidy transmitter HMW90. HMW90 is designed to measure relative humidity and temperature in indoor environments, focusing on high accuracy, stability and reliability. HMW90 can calculate many parameters from the initial values, such as dew point, mixing ratio and absolute humidity. [1,1.]

XMW90 is one of the newest devices studied in this project, and its software is clear and organized. The programming language used in XMW90 software is C++ and it uses a commercial EmbOS operating system.

3.1.2 HMT330

HMT330 is a humidity and temperature transmitter series for industrial applications focusing on wide customization and stable measurement. HMT330 software is programmed in C and it uses Tiny Simple Fast Un-Commercial Kernel (TSFUCK), which is a Vaisala proprietary non-commercial operating system. [2,1.]

3.1.3 DMT143

DMT143 is a miniature dew point transmitter that can be installed directly into pressurized systems. The sensor performs well long-term and is very resistive to

different kinds of conditions such as getting wet. DMT143 is programmed using C and it uses the TSFUCK operating system. [3,1.]

3.1.4 HMP155

HMP155 is a humidity and temperature probe for reliable measurement. The probe is designed to be very resistant against different conditions. The software of HMP155 is in C and it uses the TSFUCK operating system. [4,1.]

3.1.5 MI70

MI70 is a generic piece of software used in different hand-held meters such as the humidity and temperature meter HM70. The hand-held meter is designed for spotchecking humidity and temperature, and it is also ideal for calibration of other humidity instruments. The MI70 software is done in C and it uses the TSFUCK operating system. [5,1.]

3.2 Software Components

To limit the project to a small scale test, two software functionalities were chosen for this project. This was to get a view on the difficulty of making a complete library. Because of this, the first criterion for the software components was that the estimated implementation processes of the components had to be of variable difficulty. Secondly, both components needed to have a clear outline about the universal implementation to speed up the process. Thirdly, the usefulness of the universal components was considered.

The following software components were selected by Vaisala designers for further analysis, and two of them were selected for this thesis project. The analysis of the components and the rationale behind the selection are presented in the following chapters.

3.2.1 Non-volatile Parameter Handling

Most devices require that some variables are stored in non-volatile memory, so that some information persists over shutdowns. Since this data needs to be preserved even in case of power-outages and system failures, the handling process needs to be implemented with care.

Non-volatile information handling is mostly device-independent and could be formed in a standard library to save working time and to improve safety and quality of parameter handling, so that information is not lost in error situations and new parameters can be added efficiently.

Almost every device, and each device in the scope of this project, needs parameter handling, and since doing it right is both important and time-consuming, parameter handling as a universal library component would be very useful.

Vaisala has a draft standardization document defining the parameter handling, so based on this, the definition could be finished and built into a library component. One of the devices, XMW90, has a parameter handling implementation based on this document, so implementing parameter handling could be, while time-consuming, quite straight-forward.

3.2.2 Command Interface

All the devices in the scope of the project can be commanded via a serial port with the UNICOM command script language. Standardization of the command parsing process and defined interfaces for different commands could be useful in decreasing the amount of work needed in software creation. However, some designers questioned the usefulness of such a universal component, since the device variations of the UNICOM command language, and the interpreter would have to be tailored for each device independently.

3.2.3 Error Codes and Logging

Currently, systematic error logging is not included in every device, and error codes can be inconsistent, despite the fact that some industries can have the need to know

exactly the error history of the measurement devices. Because of this, standardization of error codes and logging of them could be useful. Difficulties include the difficulty of defining all possible error types, but this could be circumvented by implementing just the logging functionality.

3.2.4 Humidity Calculation

The standardization of humidity calculation has the concrete purpose that all the devices would yield the same results. Currently, calculation formulas may vary and two devices may give slightly different results with the same input. While some devices may have specialized versions for efficiency purposes, most devices should be able to use the same formulas, and the special cases can be easily included in the component. All in all, implementation of the humidity calculation library component would be very useful and relatively easy.

3.2.5 Miscellaneus Protocols

Information transfer protocols are a good generic example of an easily standardizable component. The implementation of a protocol can be time-consuming, but usually the device-independent software, the protocol stack, is a large part of the work, and if implemented as a library, it could be used in all future devices.

3.3 Decision

Based on the analysis above, the two software components selected were non-volatile parameter handling and humidity calculations. These are included in almost all of the example devices, and they should be very useful components. Furthermore, both have reasonably new implementations on XMW90, so starting the analysis was estimated to be quite simple.

4 Software Analysis

4.1 Humidity Calculations

In XMW90, humidity calculations define a Humicalc object class to contain values of humidity, pressure and temperature so additional values such as dew point and saturation pressure can be calculated. To accomplish this, two additional classes are defined: Gas, which contains information about the gas from which the measurements are taken, and Func, which defines a model for a single-input function. This is needed in some calculations to determine the zeros of the function; for this, the library uses three different functions to optimize the calculations. The mathematical basis of these functions is not important for this project so they are not documented here.

The correct use of the library is not documented but it can be discovered that only certain functions are called from outside the library. Thus, it can be said that those functions are the user interface of the library and all the others are just to accomplish the functionality. Based on this information-gathering method, it can be said that the correct use of the library is to create a Humicalc object, set values for it, and call the needed methods. This implementation seems to be quite modular because new functions do not need to input all the data, and because the Func implementation allows for easy search of zero-points of a function.

Weaknesses in the code consist of incomplete comments and documentation. Furthermore, some class implementations are hard to follow, mainly because of the split to source code and header files. Thus, the definition of user interface and proper documentation will be the most important improvements that the XMW90 software needs.

A general challenge in the humidity calculations is the need for different versions of a same function. For example, a function called pws calculates the saturation pressure in a given temperature. In XMW90, the calculation is done with a simple polynomial model, except that every 20 degree temperature fork has its own model. In contrast, HMT330 has a single model over the whole temperature range, and its model is much

more complicated. After testing, it could be said that both of these implementations have their benefits; the XMW90 implementation performs several times faster while HMT330 takes less codespace. The accuracy of calculations is much harder to evaluate; the results seem to differ in the magnitude of 10^-3.

XMW90 has multiple ways of calculating dew and frostpoints. The guess_tdf-function is documented to provide an estimation, while the tdfp-function should provide a better result. The latter uses an algorithm to find a function zero-point, using a value provided by the guess_tdf-function as a starting point.

XMW90 seems to lack calculation of relative humidity from capacitance, a feature which can be found in other devices. The possibility of implementing this feature in the common library was discussed with Vaisala software developers, but was dismissed because of a high amount of calibration parameters needed for the calculation.

The programmer interface is not made clear; while data seems to be set to a global Humicalc class by the RH, T and P functions, the calculations are accessed by two methods: direct function calls, and by calling functions metric() and nonmetric(), which take a constant value that presents the desired quantity as a parameter, and then call the low-level functions to get the data and then convert it if needed.

XMW90 uses C++ as the programming language while other devices use C. This is important in humidity calculations, because implementation in XMW90 is strongly based on the Humicalc object class, which is useful because the measurement data is only entered once to an object, and then all the methods have access to it. This is in contrast to the more traditional approach where each function is individually given the needed data. Because of this difference, decision about the programming language has to be made.

While C++ makes the use of code easier, there are more issues to consider, the most important one being software compatibility. C++ as a language is almost fully backwards compatible, meaning that C code can be used in a C++ program, making C++ a good choice because the existing C functions can be used as they are, or with minor changes, to maintain the object-oriented modularity in the software. However,

this same quality can be used as an argument for C: if the library is made with C it can be used as it is in C++ programs.

Due to the well planned object model of XMW90, C++ was chosen as the programming language for this project. However, it should be quite straightforward to convert the calculation functions to C if needed.

4.2 Parameters

As with the humidity calculations, the first non-volatile parameter handling software component that was analyzed was from XMW90. In the following paragraphs the results of the analysis are described.

Parameter blocks are stored in Flash. There are two parameter blocks to minimize erasing needs, since Flash has a limited amount of erase cycles. The blocks are stored in defined locations, and the parameters can be accessed via a lookup table (LUT). The LUT is genenerated on the start-up and located in RAM.

The two-block parameter implementation is done in the following way: first, the block with a larger write cycle count is selected as the parameter table, and the LUT is generated from the parameters. When parameters need to be updated, they are written at the bottom of the table and the LUT is updated. When the block is running out of space, a swap is performed: the parameters are transferred to the other table, the LUT is updated and the previous table is erased.

ParameterData.cpp contains predefined information about the default parameters, such as default, minimum and maximum values. While these are not useful in the library, the file can be used as a base for entering the default parameter data.

Parameter handling in XMW90 contains the following class structures: The Parameter Class provides functions for parameter input and output. The ParameterItem Class provides a model for parameter data. The ParameterTable class provides a model for data tables and the LUT. The ParameterTypes class contains type definitions for all data used in the software component.

The most serious limitation of the XMW90 software seems to be that parameters have to be in the universal address space. It remains unclear if the library is needed in other circumstances, and if it is, what these circumstances would be. However, the situation could be solved with the writing method implemented as a virtual function, and it is left to the programmer to implement it. The use of device-dependent functions is almost non-existent, save for instructions to allow and disallow writing (these can be included in the interfaces, and if the memory type does not require these, they can be implemented as empty functions).

4.2.1 Comparison of Devices

In DMT143, parameters are generated to the non-volatile memory from a definition file. This causes them to be in the normal address space, so the XMW90 software should function in DMT143. Unlike in XMW90, a lot of functionality in DMT143 focuses on different types of input for the parameters. This seems to be unmodular design, since type-conversions could be done at higher level. In XMW90, all data is treated as string when saved in memory, and metadata contains information about the data-types. In DMT143, it seems that command-parsing is partially integrated to parameter-handling, which is a non-modular solution. Metadata in DMT143 seems to be the same as in XMW90. DMT143 does not use a LUT because the parameters are in the memory in a specific order.

In HMT330 data is stored in separate address space accessed by functions. This could be solved in the general library by separating the actual writing functionality to a separate class which would have to be implemented in the application level.

4.2.2 EEPROM and Flash

One of the design issues to be solved is that EEPROM and Flash memory need different handling. XMW90 parameter implementation is designed strictly for Flash, and there are multiple ways to accommodate it for EEPROM. The simplest way is to keep the data format exactly the same, using two data blocks for the parameters and writing to them as when writing to Flash. The downside of this is that this uses twice as much memory as is needed for the parameters. Thus, an alternate solution would be to modify the memory accessing functions so that data is directly inserted to the right

place in the memory, keeping the amount of memory used constant and making the LUT and second parameter block needless. However, this would require a lot of work with the library.

5 Implementation and Testing

5.1 Definitions

For this section, the following definitions are used. “Library level” will refer to the reusable software produced by this thesis project, and will be common for all the devices. “Library component” refers to a specific part of the library, like humidity calculations or non-volatile parameter handling. “Application level” will refer to the device-specific software that uses the library, and “user” will be the programmer implementing the application level and using the library.

5.2 Humidity Calculations

5.2.1 Correct Use

Based on the analysis of the XMW90 source code in chapter 4.1, the structure of the library component is determined as presented in Error! Reference source not found., with Humicalc class marked in red. The correct use of the library is as follows: a Humicalc object is created and the measurement data (temperature, pressure and relative humidity) is entered to it via appropriate functions. Then, the metric and nonmetric functions can be used to calculate and output data from the Humicalc object.

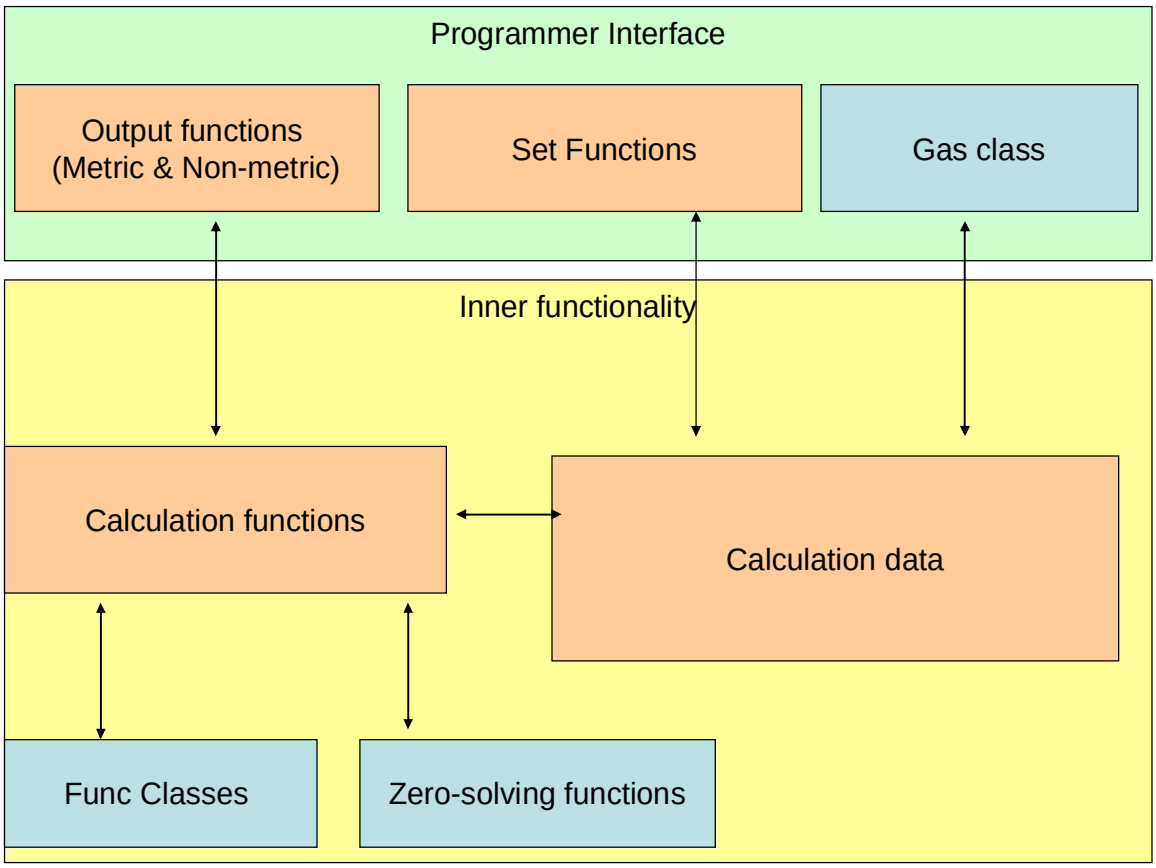

Figure 2 - Structure of the library component for humidity calculations

The programming interface will be defined as presented in Figure 2: the Humicalc class functions RH, T and P will be used to set values for, respectively, relative humidity, temperature and pressure. All output will be done with the functions metric, nonmetric, and si and q if needed; these functions require an indentifier of the quantity needed (listed in convert.cpp) and will return the desired value in desired unit. This definition will make the interface to Humicalc class simple and easy to control. Since the software will be used in embedded systems, the Humicalc object itself is created as a global object so that all functionalities have access to it.

5.2.2 Updating the Library

When creating new functionalities for the library, the interfaces defined in the previous chapter should be maintained. That is to say, new methods should be added to the

Humicalc class, so that they use the Humicalc object data instead of call parameters. This has a number of benefits: a Humicalc class is created so that it recalculates values only when one of the fundamental parameters is changed; this way, methods can be efficiently called each time values are needed, so the programmer does not have to worry whether the values have changed. Also, using the functions is simpler without a large amount of parameters; this is simply an advantage of the object model.

Due to the multi-class structure of the library and after a failed test, it was decided not to try to make all the possible functions private, since they are used by majority of the classes, and this kind of abstraction would do more harm than good.

5.2.3 Contents of the Library Component

The calculation functions to be included in the programmer interface will consist of XMW90 functions which are called outside the library file; this is because XMW90 seems to have nearly the same functions as the other devices, with next to no differences. Some functions can have multiple implementations due to the specialization need of embedded devices (space versus calculation power), with the desired version selected with a #define. While the device library lacks functions to calculate relative humidity from sensor capacitance, these will not be imported from other devices since the implementations seem to rely on sensor data which is usually not located in the library.

5.2.4 Testing

Test Conditions

The implemented library component was tested in a 32-bit Windows PC environment using the GNU g++ compiler version 4.6.3 and the CodeBlocks programming environment version 10.04. This required some changes that are all documented here, and none of them should have any effect on the performance of the software on Vaisala's IAR cross compilers, which are used to compile the actual product binaries.

Based on the definition in the previous chapter, the library was formed and tested according to the test conditions stated above. The main problem with the testing is compiling. This consists of two main problems: overloading and scope calls. For some reason, the library seemed to be unable to reach some calculation functions defined in the cmath standard library, despite the fact it was included in the file. The other problem was that the library uses overloading, that is, using the same function name with a different type and amount of parameters to create different functions. For some reason, the compiler had trouble understanding this concept. At this point, the testing was postponed until information about the compiler was gathered.

The library seems to have two major problems related to compiling, mainly caused by definitions of FLT_EVAL_METHOD and GLIBCXX_USE_C99_MATH. The first definition is done by the preprocessor and it selects types for keywords float_t and double_t, which are used in the library to enable greater portability. For some values of FLT_EVAL_METHOD, keywords float_t and double_t are defined to be of the same type, causing errors with functions overloaded with these types. When investigating this, there seemed to be no simple method to set FLT_EVAL_METHOD value for the preprocessor, so this issue was resolved by setting the value in source code. This solution is for testing only, and the definition will be removed for the final version.

C and C++ languages have separate standard libraries; for example, the library file providing math functionality is math.h in C and cmath in C++. Cmath includes math.h but removes definitions for certain macros, like the functions isfinite and isnormal that are used in the project, if a certain statement, GLIBCXX_USE_C99_MATH, is true. Then cmath redefines these macros inside std-namespace. The problem using this namespace is that with the compilers used in Vaisala, the namespace is not used, causing compatibility errors with the software. To resolve this, the value of GLIBCXX_USE_C99_MATH was changed from C++ config header.

After these changes, the library was successfully compiled in a PC environment. The details above should allow others to test the library on a PC without difficulty.

After compiling, the library seems to work fine. Validity of the calculations is not certain, so this should be tested when using this library.

5.3 Parameters

As in humidity calculations, the XMW90 source code was chosen as the basis of the parameter library component because of its good readability and emphasis on modular design.

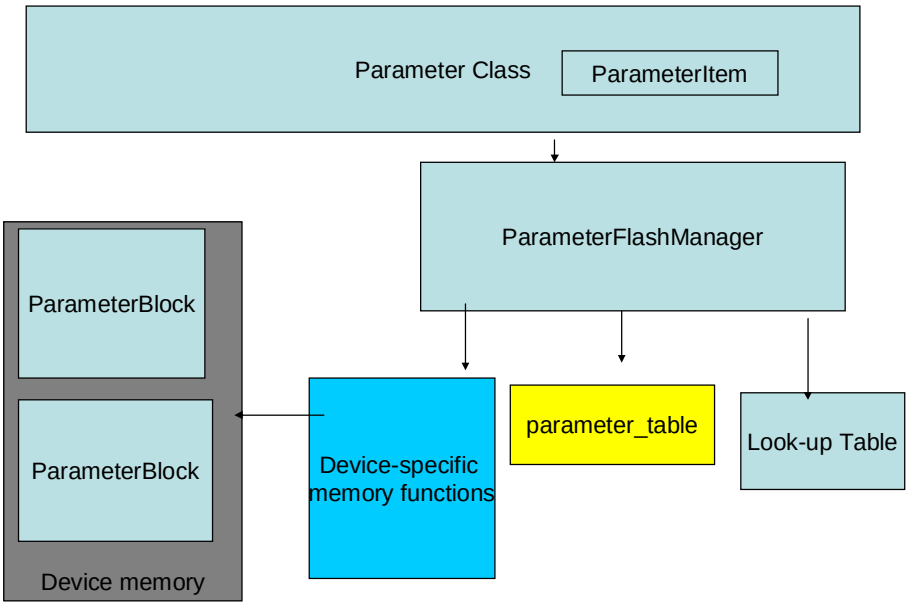

Figure 1 - Parameter structure

As seen in Figure 1, the main programmer interface is the Parameter class, which is used to access parameters in the memory. The memory structure is designed for Flash to minimize write cycles but should also be usable on EEPROM. To allow use in both normal and separate address space, memory access functions will be written as virtual functions, to be implemented by the programmer. Also, the programmer needs to write a data structure named parameter_table, which will contain the ParameterItem structures for all device parameters. This will be used by the ParameterFlashManager to check if a parameter exists. The programmer will also make Parameter objects for each ParameterItem; these are used to access parameter data and metadata.

Since improving on EEPROM efficiency would basically mean rewriting ParameterFlashManager.cpp file (discussed in 4.2.2), parameters will be saved in EEPROM using the same format as with Flash. In case a device using the library does

not have enough memory, then a more EEPROM-friendly version of the library should be created. This, however, was left outside the scope of this project.

5.3.1 Implementation

The most important modification to be done was to remove the memory dependency of the component. This was done in two parts: First, the memory dependent functions had to be identified and replanned so that the programmer can easily implement them for the needed memory type. This implementation should happen in the same file as the memory sizes and addresses are set, so that there would be a single memory definition file to be written by the programmer. After this, the memory dependent code was successfully replaced from the source code.

In XMW90, the memory access is done with a direct memory referencing, since the flash memory is defined to be part of the normal address space. However, since this is not always possible, the references had to be replaced with functions that either are directly defined by the programmer or they have to call functions defined by the programmer.

The memory functionality described is mostly based on the current implementation of the XMW90 library, so that the library can function with as small changes as possible. These functions include reading a single byte, reading and writing a larger space, enabling and disabling writing, setting and getting the flash header and getting the data table address. To limit the amount of needed functions, all the memory access functions use relative memory beginning from the data table of a memory block, and this is resolved inside the functionality into absolute memory address.

While implementing the memory independence, a problem related to the design philosophy emerged. When writing in C++, the correct way to implement an interface that is to be implemented by the programmer is to create an object class and to create a set of virtual functions that the programmer has to overload later with a derived class. The problem with this approach is that ParameterFlashManager is not used at the programmer interface, so it would require changes to the architechture to allow setting the memory functions. Also, it is used as a static class, so the constructor cannot be used to set the memory handler. Secondly, most of the functionality

described in the last paragraph can be done without objects, so using objects might cause needless work. However, despite it not being part of the programmer interface, the function ParameterFlashManager.initialize is called at the programming level, the main.cpp file to be exact. Because of this, the initialize function could be used to pass the memory-specific read-/write-object to the Flash Manager to be stored in a static variable.

The alternative is to introduce the functions in a normal library file, but then instruct the programmer to implement these functions in a specific file. This file could be also used to set the memory addresses; this is currently done in the parameter handler, which is not good since they need to be set specifically for the memory used. This approach would retain the XMW90 software architecture, but might be considered to be an obsolete solution more fitting for the C-style programming, not for the object-oriented architecture.

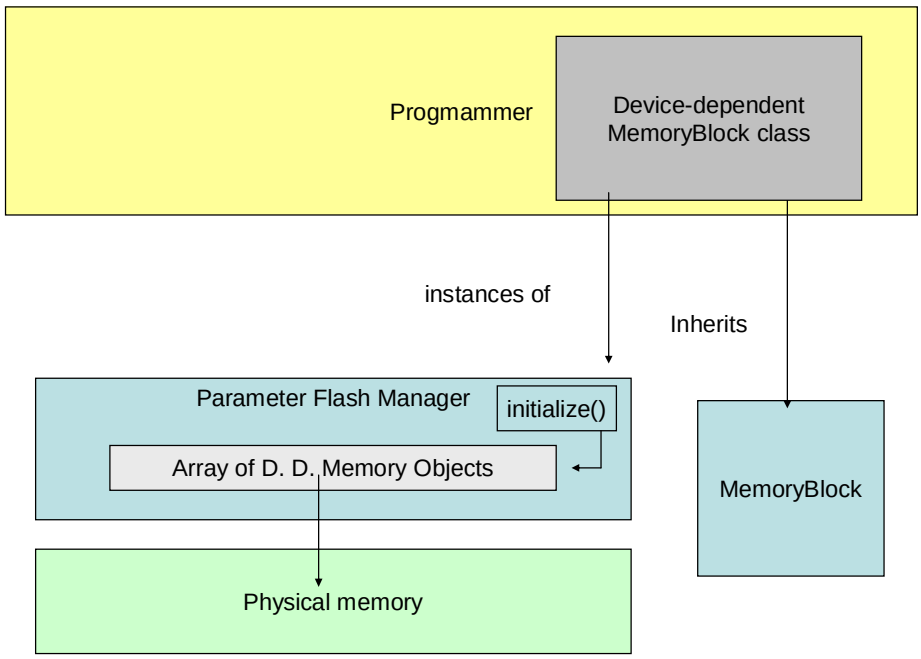

Figure 2 - Device independent memory handling

After careful analysis of the situation, the problem was resolved as Figure 2 illustrates. The memory functionality was formed into a class called MemoryBlock, which consists of information about the memory block, and functions to write and read memory. Most of these functions were pre-implemented. The programmer then creates a class which inherits the MemoryBlock, with functions _writeByte, _readByte,

enableWrite and disableWrite implemented, and passes objects of this type to the Parameter Flash Manager via the initialize function. The Flash Manager stores these objects to static variables and uses them one at a time to access a particular block.

As it was established in the previous paragraph, the memory functionality was abstracted into four device-dependent functions, which are _writeByte(int address, char byte), _readByte(int address), enableWrite() and disableWrite(). WriteByte writes the desired byte in the desired memory location; this function is then used by other functions in the class to allow writing according to an offset of a data table, or to allow writing a large amount of data at once. ReadByte functions similarly. EnableWrite and DisableWrite are included in case the memory type requires some actions before reading and writing; if not, these can be left empty.

5.3.2 Correct Use

The structure of the class Parameter is represented in Figure 5. The class is used to access the permanent memory. When a parameter has to be accessed, a parameter object has to be created with a parameter item as a constructor parameter. This object can then be used to access parameter data and metadata. The only function that is part of the programming interface besides the Parameter class is the parameter flash manager initalization function, which needs to be called at start-up. This function needs to have the programmer-made MemoryBlock-object to gain access to the permanent memory.

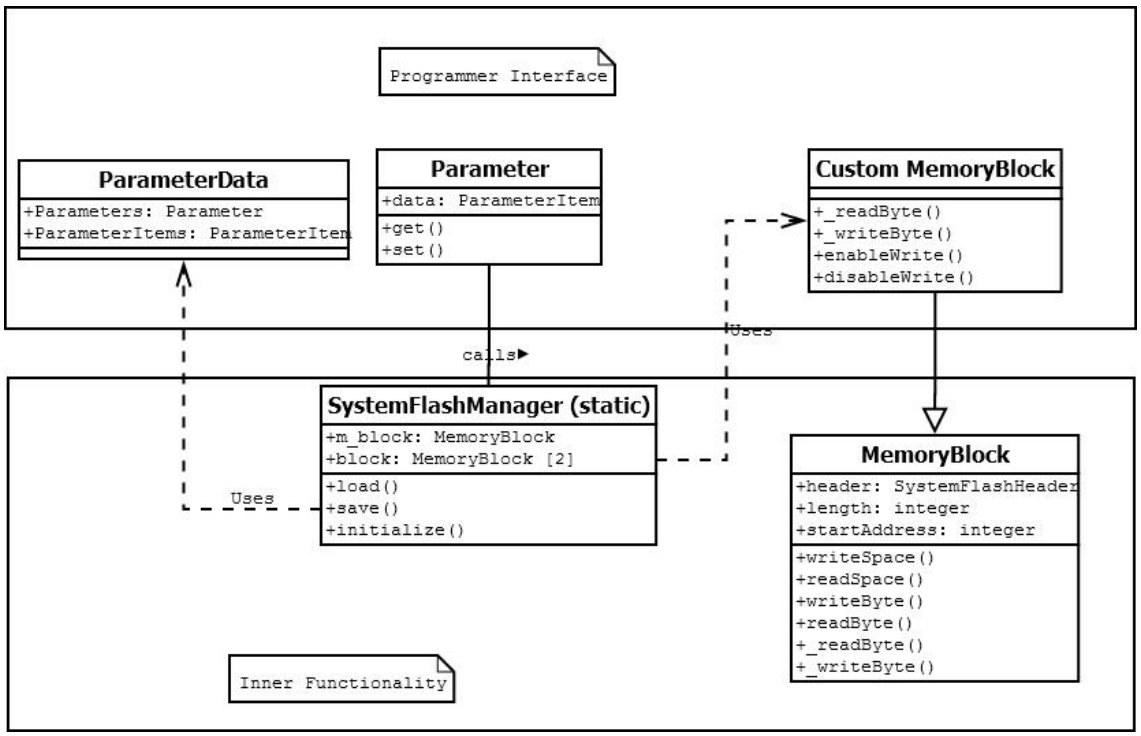

Figure 3 - Parameter class structure

ParameterData is not part of the library because it contains default values for the parameters. This needs to be set by the programmer.

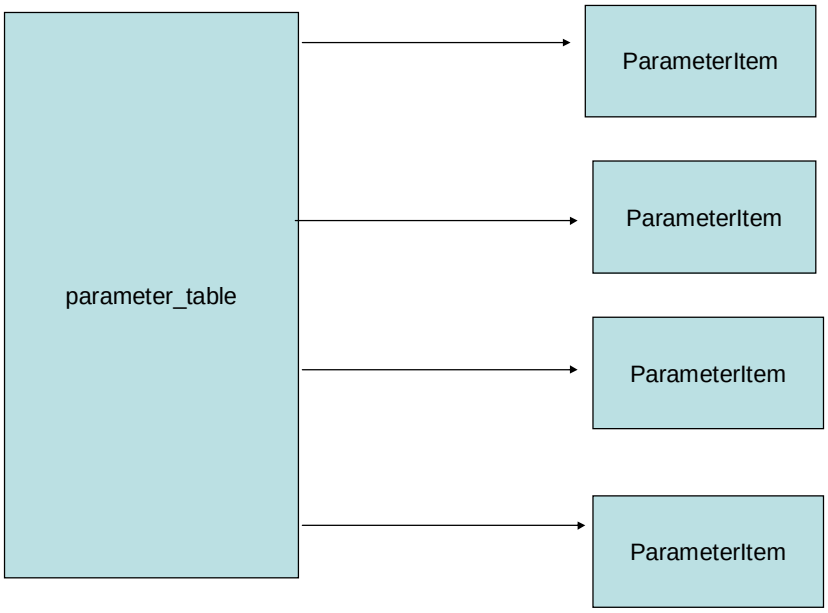

Figure 4 - Parameter metadata table

As presented in Figure 4, the programmer has to create constant ParameterItem for each parameter. The contents of ParameterItem will be explained later. Then, the programmer creates parameter_table which will point to each of the items, so it can be used by ParameterFlashManager to check if the parameters exist.

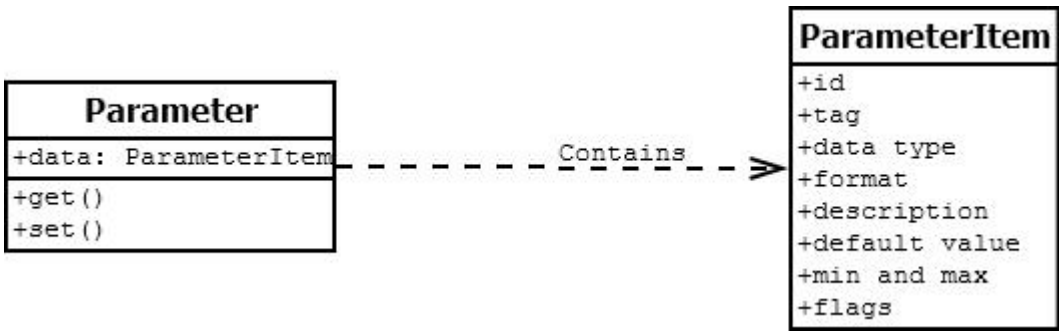

Figure 5 - Parameter and ParameterItem

Figure 5 presents the Parameter object and ParameterItem structure. As was suggested in the previous chapter, the programmer creates ParameterItem for each device parameter. Then, a Parameter object must be created for each ParameterItem. The object is then used to access and save the parameter data.

As discussed in 5.3.1, the user needs to implement a class derived from MemoryBlock, with the functions _writeByte, _readByte, enableWrite and disableWrite. When using memory in normally addressable memory space, the implementation can be very straightforward: _writeByte writes the parameter byte to the parameter address, while _readByte returns a read byte from the parameter address. EnableWrite and disableWrite can be implemented when needed.

The MemoryBlock class was programmed to use 16-bit addresses. These can be replaced with a type defined in a separate definition file in a future version of this library, but in the current version the MemoryBlock can be used in a 32-bit system by keeping the whole address in two variables and handling the less significant variable to the default MemoryBlock constructor. Then, _readByte and _writeByte can assemble the whole address and check if the low address rolls over and then they just increment the more significant variable by one.

Another possible issue is that it may be faster to write several bytes simultaneously rather than writing a single byte. This can be accomplished in the limits of the interface. For example, enableWrite can be used to create a data buffer, and _writeByte would add data to the buffer instead of directly writing it. Then, disableWrite would write the whole buffer to the memory using the most efficient way possible.

5.3.3 Testing

Test Conditions

The parameter library component was tested using the same test conditions as was used in humidity calculations library component testing (specifications found in 5.2.4). The primary test case was to implement a class derived from MemoryBlock, so that the class would read and write to the addressable memory space, and test this by writing a parameter value and reading it. As when testing the humidity calculations, it was assumed that most of the functionality is unaffected by changes and is not in need of testing.

Changes

While testing the library component, two compiler-dependent problems emerged. First, the parameter metadata initialization in the original implementation uses a structure called designated initializer, in which a struct is initialized with a list containing the names of the variables. This is supported by GNU C compiler but not the C++ version. Since the structure in question contains other structures and unions, the initialization with exactly the same result (the contents stored in flash) may be impossible. However, since this is not part of the library, it is not a problem. For this test case, a function was made for the initialization purpose and then called in the main.cpp file.

The other problem involved the Flash header that is written for each memory block. The header is a structure consisting of a union of three variables (two 8-bit integers and one 32-bit integer) and a byte array with a length same as these variables combined, and an integer for the CRC value. The purpose of this implementation is to enable writing the individual variables to the header and to use the array to quickly read and write the whole header. The problem with this solution is that some compilers add padding to the data stored to the memory so that it forms 32-bit blocks. This way, there were two excess bytes of data between the two bytes and the 32-bit integer, and these influenced the CRC calculation. This was solved by changing the order of the integers so that the 32-bit one is before 8-bit ones, so the padding bytes will be after the real data. This way, the array can be used with a length of six bytes so the excess bytes will never need to be considered.

Results

After the above changes and a number of bug fixes, the library was successfully tested in PC environment. The MemoryBlock class was derived and implemented as described in 5.3.2, and the interface was found to be versatile enough to support the variations discussed there. At this point, no critical problems affecting the use of the library component were detected.

6 Establishing Maintenance Policy and Version Control

There were several issues to be resolved about maintaining the library. First, there must be someone responsible for maintaining the library. Second, there must be rules on who can submit changes to the library. Third, information about library versions must be accessible by the users. Fourth, there must be rules about the compilers on which the software must compile. Last, if individual changes are needed on the library on specific devices, there need to be rules about handling these situations.

The solutions presented in this chapter were discussed with Vaisala designers and were deemed the best options. The rationale for this and the alternatives for the solutions are also discussed in this chapter.

Vaisala will assign a person who has the main responsibility for maintaining the library. This person will review suggestions for improvements and accept or deny them. The process is meant to be flexible and unofficial so that suggestions can be made easily and the library can be expanded and improved relatively easily.

Vaisala already uses Confluence to share information, so it will be used to share the documentation and version history of the library. Confluence is a wiki designed for enterprises to enable information sharing for teams and units [6]. Confluence will be used to keep the programmers up to date on version updates and current documentation.

Vaisala uses Apache Subversion for version control, so it will be used to maintain the version control of the library. Apache Subversion is anopen source version control system which is needed here to provide control over software updates and parallel versions, and to keep track of all the changes [7].

The software must work on both ARM and MCP 430 (manufactured by Texas Instruments) microcontroller IAR compilers, since these are used in Vaisala products. Also, the software must work on the GNU g++ compiler for testing purposes on PC environment.

The primary way of making customizations to the library should be a custom header file that can be edited by the programmer, and if something has to be configured, the

primary solution should be to make a general change to the library so that the header file can be used to change the behaviour if needed. If such a general change is impossible, then device-specific changes to the library are allowed, but they must be documented to confluence, and a separate Subversion branch must be made of the software.

7 Conclusion

After four months of work, the two software components were assembled and tested. This section will discuss the success of the thesis work and analyze possible ways to make the library making process faster, and to improve the components made in this thesis work.

The software components were assembled using the Vaisala software as a base, with only small changes, so they should be usable on Vaisala platforms. However, testing of this was not included in this thesis work because of the time it would take. Since the software was only tested in PC environment (details in 5.2.4 and 5.3.3), it is not certain that the library will work on Vaisala devices. Therefore, a possible follow-up project would be to test the library in different Vaisala devices and then document the deficiencies in the library. Part of this work was done in the analysis phase, but without any actual testing, it is impossible to say if the software would actually work.

The standards of success for the components were that they would work in different platforms. This task was time-consuming, so some things like fine-tuning the software architechture for the sake of clarity alone was not in the scope of the thesis. For example, the Parameter class of the parameter handling library component could be rewritten with derived classes instead of the current solution using the Template structure in C++; however, this does not affect functionality so it has been left for future projects. Similarly, the scope of methods inside classes was deemed an unimportant issue; this too can be fixed in library update projects.

The library making process could be improved. If the person making the library was one of Vaisala designers, the analysis phase would be almost completely removed, since the designer would already know the structure of the component that he/she would work on, and the main design issues when using the component on different devices. In this project the analysis phase concentrated mostly on analysing the XMW90 software, and the information gained from comparing it to the software of the other devices was modest at best, since it was very difficult to find the important design issues in the code. The use for the comparisons was mostly verifying the existence of some issues. If Vaisala decides to use trainees to expand and improve upon this library, the analysis should consist of another round of interviews, because

this way the design issues with a component can be found, and then they can be verified by checking the code.

The implementation process took approximately four months to finish, with estimated 60% work time. For a Vaisala designer, the implementation time is estimated to be less due to the reasons stated in the previous paragraph and because of overall higher experience, so it should be possible to add a single software component to the library in two to three weeks when working full-time. The total amount of components and the difficulty of individual components are impossible to estimate here, so the total time of implementation is to be resolved by Vaisala, if they decide to implement the full library.

References

1 Vaisala Oyj. Vaisala products – XMW90 datasheet [online]. 2012.
URL: http://www.vaisala.com/Vaisala%20Documents/
Brochures%20and%20Datasheets/
HMW90-Series-datasheet-B211183EN-B-LoRes.pdf.
Accessed 4 October 2012.

2 Vaisala Oyj. Vaisala products – HTM330 datasheet [online]. 2011.
URL: http://www.vaisala.com/Vaisala%20Documents/
Brochures%20and%20Datasheets/
HMT330-Series-Datasheet-B210951EN-C-LOW-v2.pdf
Accessed 4 October 2012.

3 Vaisala Oyj. Vaisala products – DMT143 datasheet [online]. 2012.
URL: http://www.vaisala.com/Vaisala%20Documents/
Brochures%20and%20Datasheets/
DMT143-Datasheet-B211207EN-B-LOW-v2.pdf.
Accessed 4 October 2012.

4 Vaisala Oyj. Vaisala products – HMP155 datasheet [online]. 2011.
URL: http://www.vaisala.com/Vaisala%20Documents/
Brochures%20and%20Datasheets/
HMP155-Datasheet-B210752EN-D-LOW-v4.pdf.
Accessed 4 October 2012.

5 Vaisala Oyj. Vaisala products – HM70 datasheet [online]. 2011.
URL: http://www.vaisala.com/Vaisala%20Documents/
Brochures%20and%20Datasheets/
HM70-Datasheet-B210435EN-C-LOW-v2.pdf.
Accessed 4 October 2012.

6 Atlassian. Atlassian Confluence – Overview [online]. 2012.
URL: http://www.atlassian.com/software/confluence/overview.
Accessed 4 October 2012.

7 Apache Software Foundation. Apache Subversion – Features [online]. 2011.
URL: http://subversion.apache.org/features.html
Accessed 4 October 2012.